#6. В определённом смысле интерлюдия.

Фокус занятия

- Эффективность труда программиста - как в этом помогает структуризация программ
- Функции и построение программ. Несколько слов о технологии программирования.
- Ориентируемся в многообразии форм реализации циклов и функций - неформально - где какая форма (или какие формы) более уместна(ы)
- Вброс в `bool` - совсем неформальный, подобно тому, как это уже было с линейной программой и if
- Может быть минимально упомянуть rune, так, вскользь
- Технический момент - преобразования числовых типов. Уже в прошлый раз мы столкнулись (при проверке, является ли число точным квадратом) с необходимостью явно преобразовывать типы: `int32` <--> `int64`, `float64` <--> `int64` и т.п.. Говорим об этом и, заодно, про округления `math.Ceil, math.Floor, math.Round, math.RoundToEven`.

Лекция

1

В основном, снова и очень не в последний раз говорим о построении, о конструировании программ. Т.е. очень хочется уже с самого начала смотреть с детьми на программирование, как на инженерную деятельность. Специфическую конечно - инженерная деятельность в любой сфере естественным образом имеет свою специфику, но именно инженерную. Т.е. речь идёт не просто о написании текстов, а о целенаправленной деятельности, и цель её - решение неких задач, и именно задачей и условиями её решения - имеющимися естественными ограничениями в средствах - и определяется эта наша деятельность.

И столь же естественным образом мы подошли к идее структуризации. Если задача достаточно сложная (а это достаточность начинается довольно быстро), мы не можем решить всю задачу сразу - просто мозг не может охватить все аспекты одномоментно. Чтобы решить задачу надо её как-то разбить на части. Понятно, что разбивать задачу на части надо не тупо механически, а как-то более содержательно, так чтобы это наше разбиение было адекватно задаче. Необходимо вычленять элементы системы, инженерной задачи в данном случае, и анализировать взаимосвязи этих элементов.

В преломлении к программированию разговор идёт, естественно, о структуризации программ. Повторяюсь, но примерно в таком ключе: программа представляет из себя сложный инженерный объект. Сложный в том смысле, что программа состоит из некоторого достаточно большого числа взаимодействующих между собой элементов/деталей/"механизмов". Кавычки у слова "механизмы" в данном случае весьма условные, вполне можно обойтись и без них.

А дальше: функции - это как раз и есть способ реализации этих вот деталей/механизмов. В самом деле, программа - это инструкция по выполнению определенных целенаправленных действий. А функция - это и есть некое законченное действие. Законченное в смысле некой своей целостности, завершенности. Скажем так, действие функции можно описать более или менее коротко, при этом связно и содержательно (ну, т.е описание типа "эта функция выполняет первые 30 действий" не есть содержательное описание). Описание должно иметь непосредственное отношение к выбранному нами, построенному нами способу структуризации задачи, а функция, естественным образом должна соответствовать построенной структуризации и адекватно встраиваться в решение проблемы, т.е. в программу.

Обязательно повторяем синтаксические моменты функций – возвращаемое значение (одно, несколько, ни одного), параметры (один, несколько, ни одного), локальные переменные.

2

Какой нибудь пример, иллюстрирующий всё вышенаписанное. Успешно впишется здесь какая-нибудь **задача с простыми числами**, например:

```
вводится натуральное четное число d;
представить его в виде разности двух простых чисел 10 способами.
```

Решаем.

- Сначала пишем главную процедуру. Что там надо делать? Перебирать последовательно (кстати, последовательно не обязательно, но так проще и заведомо не хуже) пары чисел, различающихся на d, и проверять их оба или они простые. Набросали такой цикл, не думая пока о занозе - о том, как проверять число на простоту. Может быть даже не кидаться сразу проверять простоту числа, поиграемся хоть немного с циклом из основной процедуры, отметим, например, что перебирать стоит только нечетные числа – внесём в главный цикл соответствующие поправки. Это очень полезно для детей - отвлечься, забыть, что у нас остался в тылу незакрытый вопрос. И очень хорошо, что именно вот такая структурная технология позволяет нам отложить вопрос о проверке на простоту, не думать о нём сейчас, выключить его. Потом вернёмся, но думать уже будем только о нём. Может быть поговорить, что да, код стал сложнеее (несущественно, но всё же), зато в два раза быстрее - т.е. мы меняем мозги на

эффективность, а свободные мозги - это плюс, и плюс очень важный. И вот только потом занимаемся функцией, проверяющей, является ли число простым. И пишем сначала самый «деревянный» вариант - вариант **06_1a.go** (не, ну, возможно есть и подеревяннее, но не будем их выискивать специально).

- **И вот здесь вот скажем про тип `bool`, который и есть условие**. В общем-то уже на прошлом занятии руки чесались при проверке, является ли число точным квадратом, завести `bool`. Вводим его максимально неформально, опираясь на интерпретацию `bool` как условие. Я бы даже, может быть, не стал говорить слова `true` и `false`. А именно "возвращаем некоторое условие, которое то ли выполняется, то ли нет". Это непросто, но очень не хочется, чтобы дети писали `if что-то там {b = true} else { b = false}`. Если без них никак, ну, пусть будут, но можно и без них. Тем более, что содержательный разговор о `bool` будет в этом семестре (да, будет!), а пока можно обходиться с ними более или менее формально, типа "делай вот так", но сразу обещая что сравнительно скоро с этим разберёмся. И в итоге получится вариант **06_1b.go**
- Потом вариант с перебором до N/2 (поскольку частное не может быть меньше 2). Этот вариант можно и пропустить, в любом случае этот вариант - только переходный шаг к следующему, и зависать тут по-любому не надо и нельзя.
- Потом вариант с перебором до Sqrt(N). При этом дела замечаем, что о главной программе мы уже благополучно забыли, а ведь она-то и улучшается! И получаем вариант **06_1c.go**
- А ведь в главной программе проверяются на простоту только нечетные числа. Значит при проверке простоты не надо проверять делимость на 2 и вообще на все чётные числа. Исправляем свою функцию так, чтобы проверяла делимость только на нечетные делители - на 3, на 5 и т.д.
- Но жалко функции – она ведь была такая универсальная. Добавляем в нее проверку на четность отдельно (а в ней проверку на 2). Останавливаемся на варианте **06_1d.go**. Хотя можно и попродолжать.
- А теперь, когда мы заморочили детм голову, и они совсем уже забыли о func main (по крайней мере мы к этому стремились - пишем prime, не думая про main, полностью из main выключившись; а сначала мы писали main, не думая про prime), так вот теперь можно вернуться к main. И придумать что-то типа `06_1e.go` например. Нет, это не обязательно делать, это очень по желанию, и если дети ещё живы, но такая возможность вывернуть ситуацию есть.

По ходу всей этой истории не перестаём играть с циклами, применяя тот или иной вариант и выбирая более адекватный, объясняем свой выбор - это один из важных моментов занятия.

3

Один из существенных моментов этого занятия - поиграться с разнообразием возможностей реализации функций. И на прошлом занятии и на этом несколько в стороне оказались функции, ничего не возвращающие (то, что в Паскале называется процедурами, а в C - void function). Да, весьма часто такие функции связаны с изменением переданных переменных, т.е. с передачей адреса, а мы пока этого избегаем. Ну, и будем пока избегать, примеров такого рода функций всё равно предостаточно. Вот и поговорим о них, акцентируясь именно на смысле таких функций. Т.е. функция, которая ничего не возвращает, но что-то же она делает. Ну, вот так и есть - такие функция что-то делают, а не вычисляют. Т.е. это как раз признак такого рода функций. Вот, например, был пример `05_3.go` - обратный отсчёт, имеенно отсчитывает, т.е. делает какое-то действие. Или печатает что-то, задерживает выполнение программы на какое-то время (time.Sleep из `05_2.go`), считывает что-то с клавиатуры (fmt.Scan), что-нибудь делает с файлом - удаляет, переименовывает, переносит или копирует, и т.д. Поиграем с примерами на печатание. Примеры и для лекции, и для практики, и для самостоятельных занятий:

- вывести все простые числа от 1 до N (или от A до B – тогда у функции два параметра),
- картинки из цифр:

```
1         1         1      1111      1111
22        22       222      222      2222
333       333     33333     33        3333
4444      22        222     4
.....     1           1
```

В первых четырёх картинках один параметр - целое число от 1 до 9, в пятой - два параметра: высота - число от 1 до 9, и ширина - просто целое положительное число, всё-таки не очень большое, чтоб влезало в экран без геморроя

- первые четыре картинки из предыдущего пункта можно рисовать звёздочками, крестиками или плюсиками, например. Тогда ограничение "от 1 до 9" можно отбросить
- ещё картинки из плюсиков, крестиков и звёздочек:

```
 xxxx        x
 xxxx       x x
 xxxx      x   x
            x x
             x

 xxxxxx     xxxxxx
 x    x     x++++x
 x    x     x++++x
 xxxxxx     xxxxxx
```

Вот давайте к примеру рассмотрим две последние картинки. И там естественно выплывает функция, которая печатает одну строку. Точнее, две функции: одна печатает верхнюю и нижнюю строки, а другая - все промежуточные. И таки образом мы легко и элегантно уёдём от вложенных циклов, которые пока прибережём. Поговорим о них в другой раз. Это не значит, что надо их целенаправленно избегать, если они всплывут по ходу занятия, так пусть всплывают, тогда и скажем о них что-нибудь минимально необходимое, но самому в ту сторону лучше не заглядываться.

Первым делом берём пример `06_2a.go`, который рисует "пустой" прямоугольник (да, там не проверяются входные данные на корректность, но не в этом пафос данной серии примеров).

Понятно, что для рисования заполненного прямоугольника надо изменить в одном месте пробел на плюс. И тут начинает со всей отчётливостью выглядывать тень символов, рун, говоря на Go. В принципе, этот момент опционален, если занятие получится перегруженным - такое может запросто случиться, несмотря на то, что фактического материала рассматривается не так уж много, - то можно про руны не упоминать вовсе, но можно и упомянуть, но именно так, вскользь, вот в данном примере, просто нужен тип, чтобы передать символ, которым рисуем. Разве что обратить внимание детей, что в примере `06_2a.go` мы печатаем "x" и " " в кавычках - а это строка, а вот в примере `06_2b.go` передаём в соответствующие функции символы в апострофах, потому, что они руны. В итоге получим `06_2b.go`, который рисует пустой прямоугольник, а символ, которым рисуем границу, передаётся при вызове.

И тогда совершенно естественно появляется идея передавать не только граничный символ, но и заполняющий - получаем пример `06_2c.go`. Тут, конечно, появляется нюанс с печатью рун - fmt.Print и fmt.Println печатают число, т.е. fmt.Print('x') и fmt.Print("x") печатают совсем разные вещи, но детально об этом точно не в этот раз.

В общем, можно на этом занятии в руны не лезть, можно ограничиться примерами с рисованием картинок из цифр. А залезем - тоже неплохо.

4

- Ну, и последнее, если влезет. С необходимостью преобразовывать типы явно все уже наверняка столкнулись, возможно и не один раз. По крайней мере на прошлом занятии в пример с проверкой на то, является ли число точным квадратом, было выражение:

```
    n2 := int(math.Sqrt(float64(n) + 0.1))
```

Пробежимся ещё раз как это работает: преобразовываем int к float64, чтобы мог сработать math.Sqrt, потом увеличиваем число на 0.1 - это во избежания вариантов из-за вычислительной погрешности получить число с дробной частью 0.9999999999999999...), а потом обратно конвертим в int, отбрасывая при этом дробную часть, чтобы сравнить квадрат полученного числа с проверяемым числом. Да, оба преобразования туда и обратно выглядят осмысленными.

- Был также пример с подсчётом среднего арифметического нескольких float64 - количество чисел, которое естественно int, при делении надо было конвертить к float64.

- Или например вот такой пример: написать функцию, которая находит расстояние между даумя целыми числами. Всё просто, расстояние между a и b равно |a-b|. И вот тут-то мы нарываемся на то, что math.Abs и получает и возвращает float64. Итого, приходится писать так:

```
func intDist( a, b int) int  {
    if a > b  {
        return a-b
    }  else  {
        return b-a
    }
}
```

Но меня здесь мучит жаба - ведь есть готовый math.Abs. Ладно, его реализация простая, легко написать самому, а если бы она была гораздо более сложная и длинная? Хочется всё-таки применить math.Abs. И применим, только при этом надо будет исполнить конвертацию туда-сюда:

```
func intDist( a, b int) int  {
    return int(math.Abs(float64(a-b)))
}
```

- Неплохо было бы здесь до кучи вкрутить округления `math.Ceil, math.Floor, math.Round и math.RoundToEven`:
- func Ceil(x float64) float64 Ceil returns the least integer value greater than or equal to x.
- func Floor(x float64) float64 Floor returns the greatest integer value less than or equal to x.
- func Round(x float64) float64 Round returns the nearest integer, rounding half away from zero.
- func RoundToEven(x float64) float64 RoundToEven returns the nearest integer, rounding ties to even.

Пусть дети сами поиграются с этими функциями и с преобразованиями int<-->float64.

Ещё один забавный пример - `6_03.go` . Программа ищет все натуральные числа k (k < 64), для которых 2^k^+ 1 делится на k. Почему k < 64 понятно - чтобы 2^k^ влезло в int64. Но вот чтобы проверить делимость n = 2^k^ +1 на k необходимо, чтобы они были одного типа, поэтому и делаем k int64.

##Практика. **Большая задача** Хорошо бы дать что-нибудь многосерийное. Только надо предварительно разобрать способ решения, хотя бы вкратце. Т.е. это довольно времязатратное развлечение. Давать ли его и, если давать, то насколько продвигаться по намеченному пути - вопрос, который каждый решает сам на местности. Хороший пример такого рода серии - серия задач про треугольные числа. *Треугольные числа* - это последовательность 1, 3, 6, 10, 15, 21, 28, ... Название объясняет рисунок:

```
 1    *
 3    **
 6    ***
10    ****
15    *****
21    ******
...
```

n-е треугольное число - это количество звёздочек в треугольнике высоты n `t[n] = 1 + 2 + 3 + ... + n = n*(n+1)/2`

И пошли задачи:

- a. вывести все треугольные числа до 1000,
- b. затем написать функцию, находящую треугольные числа суммированием (потеря в скорости!),
- c. затем с помощью явной формулы.
- d. далее – проверка числа на треугольность суммированием или,
- e. решая квадратное уравнение. В пунктах d. и e. используем bool, причём совершенно аналогично тому, как это было в примерах с проверкой числа на простоту.
- f. затем – представимо ли число в виде суммы двух треугольных,
- g. потом – в виде суммы трёх треугольных. И в f./g. тоже bool
- h. в заключение – выводить не только да/нет, но и само разбиение на сумму двух
- i. и на сумму трёх треугольных чисел.

N.B. Известно, что любое натуральное число разложимо в сумму трех треугольных, если считать 0 треугольным числом номер 0: t[0] = 0.

Упражнения. Все пункты хорошо бы делать с помощью вызова в цикле функции, которая рисует одну строку a.

```
1 2 3 4 5 6  (-числа разделяются одним пробелом-)
7 8 9 10 11
12 13 14 15
16 17 18
19 20
21
```

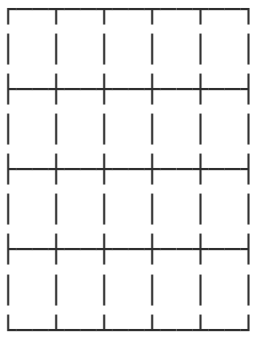
b.

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
```

c. разные треугольники из крестиков/звёдочек; вариант - те же треугольники, но "пустые" внутри

```
x              x       xxxxx       xxxxx
xx            xx       xxxx         xxxx
xxx          xxx       xxx           xxx
xxxx        xxxx       xx             xx
xxxxx      xxxxx       x               x
```

d. рисуем клетчатую таблицу символами псевдографики:



Для рисования строчек, а их здесь четыре вида, пишем соответствующие функции. Заметим, что функции достаточно одной, но ей надо передавать, кроме размеров, четыре руны: крайнюю левую, крайнюю правую, внутреннюю на краю клетки, внутреннюю внутри клетки: e. тоже клетчатую таблицу, но без псевдографики:

```
+--+--+--+
|  |  |  |
+--+--+--+
|  |  |  |
+--+--+--+
|  |  |  |
+--+--+--+
```

Тут история аналогичная, но попроще - рун в функцию передаём только две: внутри и на краю клетки.

Задачи.

1. Разложить четное число, большее 4 , на сумму двух простых – все способы.
2. Найти все решения в целых числах уравнения a. x^2^ – xy + y^2^ = N b. x^3^ + y^3^ = N c. sqrt(x) + sqrt(y) = sqrt(N) d. 2x^2^ – 5xy + 2y^2^ = N e. x(N + x) = y^2^
3. Найти N-е число Фибоначчи, сумму первых N чисел Фибоначчи.
4. a. Найти все представления числа в виде суммы трех квадратов. b. Найти все представления числа в виде суммы трех кубов.
5. Найти все четырех-(пяти-)значные квадраты, запись которых состоит только из четных цифр.
6. Найти все пары простых чисел-близнецов (т.е. простых чисел, отличающихся на 2), не превосходящие 30000.
7. Найти остаток от деления (N-1)! на N (критерий Вильсона: N – простое тогда и только тогда, когда этот остаток равен N-1).
8. Найти все числа, не превосходящие 30000, такие, что (N-1)!+1 делится на N^2 (см. предыдущую задачу – достаточно проверять только простые числа N).
9. Найти все такие натуральные N, меньшие 30000, что среди чисел от N до N+9 имеется четыре простых числа (первые такие числа – 3, 11, 101).
10. Найти первые (и вторые) N подряд идущих составных чисел.